V de Big Data

Revolucionando el procesamiento de datos masivos

Máster en Sistemas Informáticos Avanzados
Junio de 2017

Autor:

Xabier Zabala Barandiaran

Supervisores:

German Rigau i Claramunt
UPV/EHU

Iñigo Etxabe
Datik Información Inteligente S.L.

Agradecimientos

En primer lugar, quisiera expresar mi gratitud a las personas que han posibilitado la concepción y el desarrollo de la presente tesina. Agradezco a German Rigau i Claramunt, director del proyecto por parte de la UPV/EHU, el asesoramiento ofrecido durante el transcurso del mismo. Doy también las gracias a Iñigo Etxabe, cofundador y CTO de Datik Información Inteligente, por la total confianza mostrada en mi labor y el exquisito trato ofrecido desde el primer día.

Agradecer, cómo no, a mis padres José Javier Zabala y María Pilar Barandiaran el esfuerzo desempeñado para allanar, en la medida que les ha sido posible, el camino que he recorrido hasta el día de hoy. Me congratula haber sabido responder satisfactoriamente a la confianza que han depositado en mí. Por todo lo que suponen para mí, un beso enorme para los dos.

No quisiera olvidarme de aquellas personas que me han acompañado durante este maravilloso periplo. Doy las gracias a los amigos de siempre por el apoyo incondicional mostrado desde la distancia y a la gente que he tenido la fortuna de conocer durante los años de universidad. Quisiera agradecer especialmente a las personas que en este breve periodo se han ganado a pulso el privilegio a ser parte importante de lo que me resta de existencia, a los cuales me atrevo a mencionar aún con el temor de dejarme a alguna en el tintero: Adrián Núñez, Eider Irigoyen, Jaime Altuna, Jokin Etxeberria, Marta García, Mikel Etxeberria y Uxue Arostegui. No hallo palabras para expresar la gratitud que siento hacia vosotras.

Por último, pero no por ello menos importantes, quisiera evocar a todos los docentes que han tomado parte en mi formación desde aquel Septiembre del 2009 y agradecerles el esfuerzo invertido en mi persona durante estos años.

Gracias de todo corazón a la gente mencionada en este breve capítulo por haber hecho de mí un mejor profesional y una mejor persona, además de darme las fuerzas necesarias para seguir evolucionando en ambos aspectos de cara al futuro.

Resumen

Proyecto Final del Máster en Sistemas Informáticos Avanzados. Estudio empírico sobre el rendimiento ofrecido por varias tecnologías emergentes en el campo del Big Data en comparación a una base de datos tradicional a la hora de operar en escenarios que requieren un almacenamiento y procesamiento eficaz de volúmenes masivos de datos.

Dos entornos de pruebas totalmente aislados han sido erigidos sobre la misma máquina física. En el primero, se ha construido un clúster compuesto por tres nodos virtuales que operan en la misma red privada. Dichos nodos han sido dotados de tecnología necesaria para el correcto funcionamiento de Apache Cassandra [10] y Apache Spark [22]. Sobre el segundo entorno, constituido por un nodo virtual de potencia equivalente al clúster ya mencionado, se ha instalado una instancia de MySQL Server. Una vez habiendo diseñado un conjunto de consultas equivalentes para ambas bases de datos, se ha procedido a poblar dichos sistemas de almacenamiento utilizando un dataset público de aproximadamente 25GB. Para finalizar, las consultas predefinidas han sido ejecutadas pudiendo así cuantificar el tiempo de respuesta ofrecido por cada entorno.

El estudio evidencia que a la hora de trabajar con volúmenes masivos de datos el binomio formado por Apache Cassandra y Apache Spark, además de ofrecer una solución totalmente escalable y tolerante a fallos, mejora sustancialmente el rendimiento ofrecido por MySQL Server. No obstante, para gozar de dichas ventajas, se antoja necesario invertir más tiempo en analizar exhaustivamente la naturaleza de los datos que se desean tratar.

Palabras Clave: Apache Cassandra, Apache Spark, Benchmark, Big Data, MySQL.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Desde Aristóteles y su libro Segundos Analíticos [1] hasta Galileo, padre de la ciencia moderna, adalides del conocimiento han proclamado que un método de investigación basado en lo empírico y en la medición, sujeto a los principios específicos de las pruebas de razonamiento es el camino para conocer la verdad.

Hoy en día, época en la que los avances tecnológicos han posibilitado observar y medir de forma exhaustiva un gran abanico de fenómenos, la ingente cantidad de datos que se genera en el proceso es, a veces, intratable mediante las tecnologías convencionales, y por ende, es imposible extraer todo el conocimiento que atesoran. El problema, lejos de atenuarse, se acrecienta con el paso del tiempo. Estudios como el realizado por McKinsey Global Institute [2] estiman que el volumen de datos que se genera crece un 40 % cada año y auguran que entre 2009 y 2020 se verá multiplicado por 44 [13].

Para lidiar con dicha problemática, en los últimos años ha irrumpido la necesidad de desarrollar nuevas metodologías y tecnologías que permitan operar eficientemente sobre masas colosales de datos, dando como resultado el nacimiento del Big Data [11].

Empresas de renombre mundial, conscientes del conocimiento que les puede reportar en diferentes facetas de su negocio, ya se han interesado en este fenómeno. De un estudio realizado entre los altos ejecutivos de las firmas que lideran el Wall Street se desprende que el 96 % tiene planeadas ciertas iniciativas relacionadas con el Big Data, y el 80 % ya tiene finalizada alguna [18].

[1] Órganon II de Aristóteles: Recopilación de obras Aristotélicas que incluye el libro Segundo Analíticos
[2] McKinsey Global Institute

1.1. Contexto

Datik Información Inteligente [3] es una empresa tecnológica perteneciente al Grupo Irizar [4] que desarrolla soluciones ITS destinadas a la gestión del trasporte, tanto ferroviario como por carretera y movilidad ciudadana.

Uno de los productos estrella de la entidad es el denominado iPanel, concentrador de información que ofrece al operador de transporte servicios de valor añadido en la gestión de la información generada por su flota. El funcionamiento del servicio se puede resumir mediante la Figura 1 de la página 2.

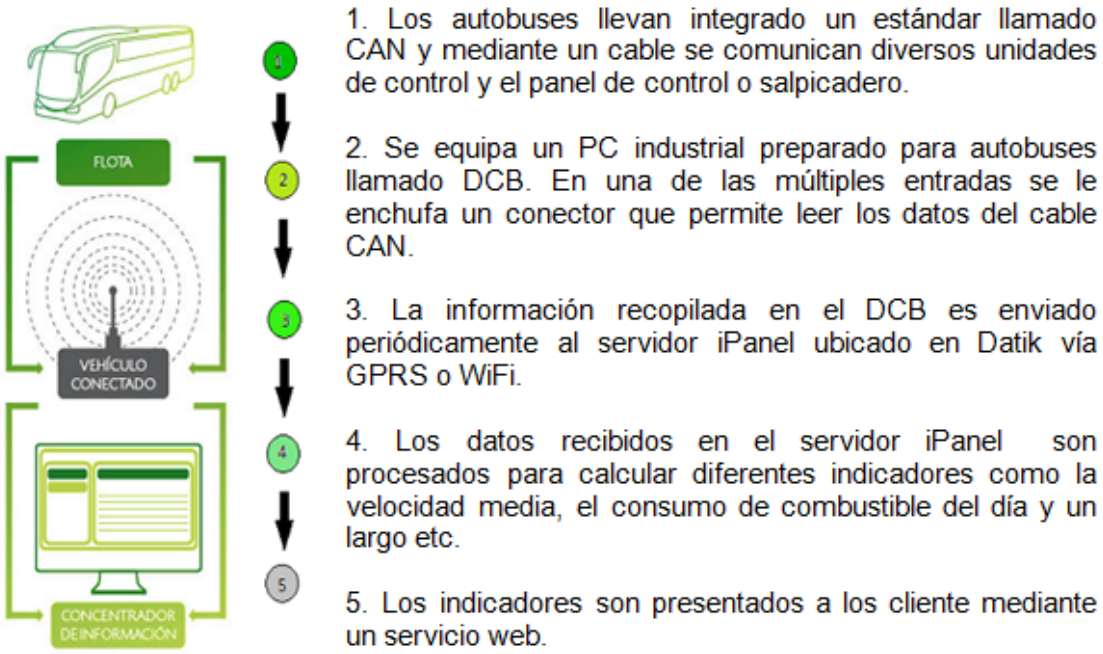


Figura 1: Funcionamiento resumido de iPanel

La incesante integración de nuevos vehículos a iPanel ha generado un crecimiento exponencial en el número de registros almacenados en ciertas tablas MySQL. Aunque el volumen actual no llega a suponer riesgo alguno para el funcionamiento del servicio, Datik tiene identificados varios escenarios en los que la situación se podría revertir, causando graves inconvenientes.

[3] `http://www.datik.es/`
[4] `http://www.irizar.com/irizar/`

Fiel reflejo de ello es el Cálculo de Indicadores: proceso de ejecución diaria que realiza operaciones aritméticas intensivas sobre datos almacenados en diversas tablas, para después, agrupar los resultados en base a diferentes criterios. Siendo dichas tablas las que mayor crecimiento experimentan, el aumento del volumen de las mismas incrementa de forma desorbitada el tiempo necesario para finalizar el cálculo.

El objetivo del presente proyecto es proponer soluciones tecnológicas que solvente permanentemente el problema del proceso Cálculo de Indicadores y que a su vez, puedan valer para lidiar con otros obstáculos de la misma índole que pudiesen emerger en un futuro.

1.2. Propuesta

Debido a que la problemática referente al Cálculo de Indicadores es originado por el incremento exponencial de datos que dicho proceso ha de tratar, no sería descabellado pensar que la solución pudiese pasar por escalar verticalmente la máquina y/o afinar la configuración del servicio MySQL ya existente. No obstante, ambas mejoras son limitadas, mientras que el volumen de los datos seguiría en aumento de forma inexorable.

Siendo imposible reconducir la situación mediante el uso de las tecnologías tradicionales, en el presente proyecto se propone realizar un cambio de paradigma y migrar a un modelo distribuido que posibilite escalar la infraestructura horizontalmente. A su vez, se sugiere dividir el problema en dos apartados, almacenamiento y procesamiento, dotando la infraestructura de tecnología adecuada para ofrecer una respuesta eficaz a cada una de las facciones.

En cuanto a almacenamiento se refiere, debido a la imperante necesidad de escarbar en hercúleos volúmenes de datos, se propone utilizar una base de datos no relacional. Dentro de la extensa y heterogénea gama de sistemas de almacenamiento NoSQL [5] existentes a día de hoy, se ha considerado idóneo el uso de Apache Cassandra [10], una base de datos distribuida del tipo clave-valor que ofrece alta disponibilidad sin un solo punto de fallo, un ratio de escrituras por segundo sustancialmente superior en comparación a sus homólogos [15] y velocidades de lectura nada desdeñables.

Por su parte, la tecnología seleccionada para el procesamiento ha sido Apache Spark [22], una plataforma de computación en clúster que ofrece una interfaz para programar operaciones paralelizadas y tolerantes a fallo que ha irrumpido con fuerza en el último lustro. A logrado desbancar tecnologías semejantes como MapReduce [4] gracias a la ca-

[5] `http://nosql-database.org/`

pacidad de almacenar en memoria los resultados intermedios del cómputo posibilitando así acelerar el procesamiento hasta 100 veces [20] en determinados escenarios.

Para cuantificar los beneficios aportados por las tecnologías propuestas se ha decidido utilizar un dataset público de aproximadamente 25 GB que recoge la información de los trayectos realizados por los taxis amarillos de Nueva York durante el año 2015[6], ya que los datos actualmente almacenados por Datik se encuentran sujetos a una cláusula de confidencialidad.

1.3. Organización del documento

El presente documento abarca el estudio empírico realizado sobre el rendimiento obtenido mediante el uso conjunto de Apache Cassandra y Apache Spark en comparación al ofrecido por MySQL Server en escenarios que requieren un almacenamiento y procesamiento eficaz de volúmenes masivos de datos.

En este primer capítulo, se ha introducido la problemática que ha impulsado la creación del proyecto y expuesto la propuesta tecnológica para dar solución a la misma.

En el capítulo 2, se desgrana el funcionamiento de las tecnologías electas para resolver el problema, haciendo especial hincapié en las peculiaridades que les permiten ofrecer una mejora sustancial en comparación a otras herramientas.

Una vez en el capítulo 3, se detalla la praxis llevada a cabo para confeccionar los dos entornos de prueba. Por un lado, se describe la construcción de un entorno centralizado compuesto por un único nodo que alberga una instancia de MySQL y por el otro, la de uno distribuido, formado por tres nodos homogéneos sobre el cual operan Apache Cassandra y Apache Spark en consonancia.

El almacenamiento y la retribución de los datos constituyen el grueso del 4to capítulo. Primero, se expone la naturaleza de los datos que conforman el dataset y se enuncian las consultas que se desean realizar sobre este. Después, se presenta el diseño de las tablas utilizadas para almacenar dichos datos, incidiendo especialmente en la correspondiente a Apache Cassandra debido a las peculiaridades que presenta.

En el capítulo 5, se publican los resultados obtenidos una vez habiendo ejecutado las consultas anteriormente definidas. Dichos resultados son representados mediante gráficas y analizados posteriormente para observar de forma nítida las diferencias existentes

[6] `http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml`

entre ambos entornos.

Para finalizar, en el capítulo 6 se presentan las conclusiones y líneas futuras para el proyecto.

Fuera de la estructura general de la memoria, se puede encontrar la bibliografía citando todas las fuentes relevantes accedidas en el devenir del presente trabajo.

Capítulo 2

Análisis de las tecnologías propuestas

2.1. Apache Cassandra

Apache Cassandra es una base de datos distribuida que permite operar sobre grandes volúmenes de datos del tipo clave/valor. Concebida en el 2008 por los ingenieros de Facebook basándose en otros sistemas de almacenamientos distribuidos como Dynamo[5] y BigTable [3], se caracteriza por ofrecer una disponibilidad total y mayor escalabilidad lineal en comparación a otras bases de datos NoSQL[13]. Para ello, se basa en un conjunto de nodos homogéneos dentro de una red en anillo que se comunican mediante un protocolo P2P de replicación asíncrona, lo cual permite realizar operaciones de baja latencia sin necesidad de un servidor maestro.

Hoy en día, es utilizada por infinidad de aplicaciones en negocios modernos, siendo la base de datos elegida por un tercio de las compañías que conforman la Fortune 100 [1]. Claro ejemplo de ello son empresas mundialmente conocidas como Apple, Facebook o NetFlix, los cuales utilizan Apache Cassandra como parte de su entramado tecnológico desde hace ya unos años. Cabe destacar que su uso no se limita al mundo empresarial. Muestra de ello es la acogida que ha tenido en el ámbito de la investigación, formando parte en experimentos punteros a nivel mundial como algunos de los realizados en el CERN [16].

[1] `http://fortune.com/2015/04/14/datastax-hp-sales-partnership/`

2.1.1. Funcionamiento

Las bases de datos NoSQL, en su mayoría, han sido concebidas para resolver un conjunto especifico de problemas. Debido a ello, para entender el funcionamiento de Cassandra cabe resaltar las peculiaridades que la distinguen de los demás sistemas de almacenamiento no relacionales.

Base de datos distribuida de alta disponibilidad

El Teorema de Brewer [7], también conocido como Teorema CAP , enuncia que un sistema de cómputo distribuido no puede garantizar simultáneamente las tres propiedades que se presentan a continuación, solo pudiendo cumplir dos de ellas al mismo tiempo, y acabar cumpliendo la restante tarde o temprano:

- **Consistencia**(Consistency): Todos los nodos ven la misma información al mismo tiempo.
- **Disponibilidad**(Availability): La garantía de que cada petición a un nodo reciba una confirmación de si ha sido o no resuelta satisfactoriamente.
- **Tolerancia al Particionado**(Partition Tolerance): El sistema sigue funcionando a pesar de que haya sido partido por un fallo de red.

Para una base de datos distribuida que promete la disponibilidad completa, el Teorema de Brewer implica la incapacidad de garantizar la consistencia total de los datos que almacena y la necesidad de esperar un tiempo indeterminado para que las réplicas de un registro modificado se actualicen correctamente.

Para lidiar con dicha restricción, Cassandra ofrece la posibilidad de afinar el nivel de consistencia [2] en cada consulta, sacrificando para ello parte de la disponibilidad.

Optimizado para escrituras

Una de las características a destacar de Cassandra es el elevado ratio de escrituras por segundo que es capaz de realizar, superando notoriamente a otras bases de datos NoSQL en este apartado [15].

Al ejecutar una consulta que desemboca en una escritura, los registros son almacenados en dos estructuras denominadas CommitLog[3] y Memtable. El primero, es un fichero

[2] https://docs.datastax.com/en/cassandra/2.1/cassandra/dml/dml_config_consistency_c.html

[3] https://wiki.apache.org/cassandra/Durability

donde se adjuntan los registros recibidos, mientras que el segundo, es una cache en memoria que apila dichos registros. Cuando ambas estructuras finalizan de almacenar los datos, Cassandra considera que la consulta ha sido ejecutado satisfactoriamente pudiendo pasar a procesar otras operaciones.

Cuando la Memtable[4] se llena, empieza un proceso de vaciado en el cual primero se dilucida el nodo destinatario de cada registro. Después, dichos registros son escritos de forma secuencial en estructuras inmutables denominadas SSTable, ficheros que representan físicamente el contenido de una tabla. Si esta operación es interrumpida, se utiliza la información almacenada en el CommitLog para recuperar las escrituras perdidas. Finalmente, cuando todos los registros han sido transferidos a su correspondiente SSTable, el CommitLog es purgado.

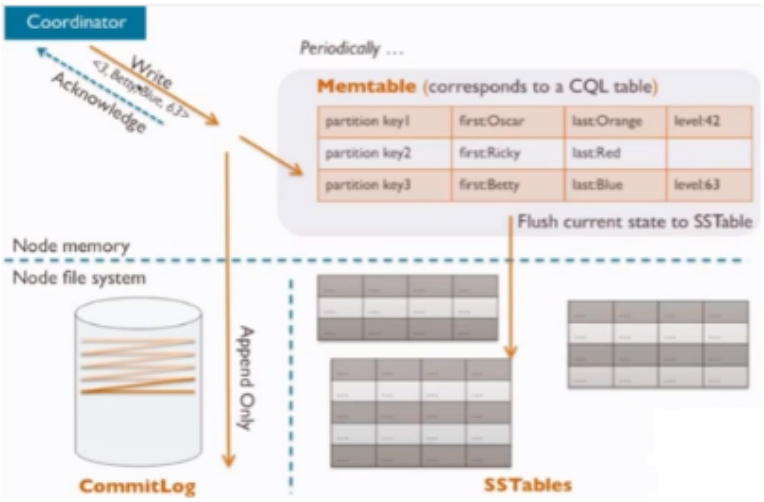


Figura 2: Almacenamiento de datos en Cassandra

Debido a la naturaleza inmutable de las SSTable, cada vez que la Memtable se vacía un nuevo fichero es generado para representar una tabla ya existente. Ello puede acarrear problemas de eficiencia a la hora de consultar la información de dicha tabla, ya que sería totalmente necesario leer todos los ficheros correspondientes. Para sobreponerse a este inconveniente, Cassandra posee un mecanismo llamado compactación que, entre otras acciones, unifica las SSTable que representan una misma tabla eliminando los ficheros restantes. El proceso recién descrito se resume mediante la figura 2 de la página 9.

[4] `https://wiki.apache.org/cassandra/MemtableSSTable`

Replicación asíncrona sin maestro

Las bases de datos distribuidas ofrecen la posibilidad de replicar los datos que se almacenan en ellas, lo cual implica que todos los nodos del sistema deban conocer el estado de cada réplica. Cassandra, gracias a un conjunto de mecanismos que se presentan a continuación, es capaz de obtener dicha información de forma local sin depender de un nodo maestro, obteniendo un clúster totalmente homogéneo.

- **Gossip Protocol**[6]: Protocolo de comunicación peer-to-peer que intercambia el estado del propio nodo y de aquellos que conoce de forma periódica.

  Cada mensaje Gossip tiene una versión asociada a él, gracias al cual el nodo que recibe dicho mensaje puede comparar con la información que posee de antemano y actualizar su conocimiento en caso de ser necesario.

- **Failure Detector**[2]: Conjunto de métodos que mediante mensajes Gossip sugieren de forma local si otro nodo del sistema esta caído o en estado transitorio.

  Dentro de la extensa gama de métodos existentes, Cassandra utiliza el Accrual Failure Detector[9]. Ofrece la posibilidad de configurar un umbral teniendo en cuenta el rendimiento de red, carga de trabajo, u otras variables. Al cotejar el valor ofrecido por el detector con dicho umbral, Cassandra es capaz de dilucidar si un nodo esta caído, y en caso afirmativo, redireccionar las peticiones a otro activo.

- **Merkle Tree**[12]: Árbol hash cuyos nodos padres más altos son, a su vez, hashes de sus respectivos hijos. La ventaja principal es que cada rama del árbol puede ser comprobada de forma independiente, sin la necesidad de descargar todo el conjunto de datos.

  La implementación de Anti-Entropia[8] en Cassandra, proceso encargado de comparar todas las réplicas de cada dato que existe en el clúster y actualizarlos a la versión más reciente, genera un Merkle Tree por cada tabla durante el proceso de compactación, los cuales se almacenan solo hasta enviarlos a los nodos vecinos. La praxis descrita implica enviar un exceso de datos por la red pero ahorra en operaciones I/O del disco local, lo cual es preferible para conjuntos de datos muy grandes.

Sin punto único de fallo

La ejecución de una consulta es considerada satisfactoria si el clúster es capaz de cumplir con los requisitos de consistencia especificados en la misma. Esta simple premisa se embarulla en un entorno real en donde los nodos de la infraestructura pueden cambiar

de estado en cualquier instante y además, seguir cumpliendo con los requisitos de consistencia, dando como resultado el no poder actualizar las réplicas que residen en los nodos temporalmente caídos.

- **Rapid Read Protection**[5]: El nodo del clúster que recibe una consulta es denominado coordinador y cumple con el cometido de identificar la máquina menos ocupada entre los que contienen una réplica del registro requerido y redirigir la consulta hacia ella.

  Si en ese preciso instante el nodo electo deja de estar disponible, la técnica Rapid Read Protection permite al coordinador escoger otro nodo entre los candidatos y redirigir la petición hacia este último, evitando así tener que devolver un mensaje de error al cliente.

- **Hinted Handoff**[6]: Técnica que almacena en una tabla interna del nodo coordinador los registros que no han podido ser replicados a otras máquinas. Al recibir un mensaje Gossip indicando que dichas máquinas vuelven a estar activas, se encarga del envío de los registros correspondientes.

2.1.2. Cassandra Query Lenguage (CQL)

El código que rige el funcionamiento interno de las consultas es generado mediante Apache Thrift[17], una interfaz para implementar llamadas a procedimientos remotos[14] que posibilita el acceso al clúster mediante diversos lenguajes de programación.

No obstante, para la gente acostumbrada a trabajar con SQL suponía un escollo familiarizarse con la sintaxis de Thrift. Con el objetivo de atraer usuarios, los desarrolladores de Cassandra optaron por añadir una capa de abstracción para así lograr un lenguaje de consultas que se asemejara sintácticamente a SQL, dando lugar al nacimiento de Cassandra Query Lenguage (CQL).

Este lenguaje de consultas presenta las mismas funcionalidades y limitaciones que el antiguo API basado en Thrift, entre las cuales destacan, entre otras muchas, la imposibilidad realizar algunas operaciones como *join*, agrupaciones y agregaciones, además de presentar diversas restricciones en la sintaxis de *where*[7].

[5] `http://www.datastax.com/dev/blog/rapid-read-protection-in-cassandra-2-0-2`
[6] `http://www.datastax.com/dev/blog/modern-hinted-handoff`
[7] `https://www.datastax.com/dev/blog/a-deep-look-to-the-cql-where-clause`

2.2. Apache Spark

Apache Spark[22] es un proyecto open source de computación en clúster. Desde el principio fue diseñado para ejecutar algoritmos iterativos en memoria sin la necesidad de almacenar en disco los resultados intermedios generados durante el proceso. Esta peculiaridad permite que los procesamientos llevados a cabo con Spark puedan llegar a ser, en algunos casos concretos, 100 veces más rápidos que los de MapReduce[4].

La base del proyecto es el denominado Spark Core. Proporciona envío distribuido de tareas, planificación y funciones básicas de entrada salida. La abstracción fundamental de programación se llama Resilient Distributed Datasets[21], una colección lógica de datos particionados a través de las máquinas que se expone mediante una API integrada en lenguajes como Java, Python y Scala.

2.2.1. Funcionamiento

Para el funcionamiento de Spark, es condición sine qua non que los nodos de la infraestructura tengan acceso a la totalidad de los datos que se desea tratar. Ello implica que para procesar un fichero de 25GB, cada nodo tiene que poseer una copia del mismo almacenado en su disco. Esta praxis es inviable, ya que más allá de los problemas de consistencia que generaría, para nada es eficiente ocupar la memoria de todos los nodos con información redundante y procesar el fichero entero cuando en realidad se va a hacer uso de una pequeña porción de dichos datos en cada ejecución.

Las bases de datos distribuidas como Cassandra solventan los problemas anteriormente mencionados. Se encargan de distribuir los datos entre diferentes nodos del clúster, ofrecen la posibilidad de acceder a ellos desde cualquier punto y mantienen la consistencia de los mismos a cambio de sufrir una pequeña latencia en el caso de requerir información almacenada en otro nodo de la infraestructura.

Tal y como se puede apreciar en la figura 3 de la página 13, al ejecutar una aplicación que opera con Spark, un componente denominado *driver* es lanzado. Debido a la necesidad de obtener recursos (CPU y memoria) para llevar a cabo la computación que se le ha encomendado, este se comunica con un nodo del clúster que, mediante especificación previa, adopta el rol de maestro. El *maestro* pregunta a todos los nodos que conforman la infraestructura sobre la cantidad de recursos disponibles que poseen con el objetivo de asignarles los *executors* correspondiente. Las máquinas que alojan al menos un *executor* pasan a denominarse *worker* y a partir de este momento, podrán comunicarse directamente con el *driver* para poder recibir las tareas que éste les envíe.

Un *executor* es una unidad de trabajo que se encarga de computar las tareas que le encomienda el *driver*. El número de *executors* que puede albergar cada *worker* está directamente relacionado con el número de recursos que este posee. Spark permite modificar ambos parámetros programáticamente permitiendo así poder amoldarse a las particularidades de cada ejecución.

Para transferir el código del programa, residente en la máquina del *driver* en formato JAR, éste adopta el rol de servidor e intenta enviar el fichero a los *workers*. Si el ejecutable que contiene el código ha sido recibido correctamente por sus destinatarios, estos responden mediante un ACK y en caso contrario, se vuelve a intentar el envío un número determinado de veces. Una vez llegado al máximo de reintentos, el *worker* que no haya enviado el ACK es considerado caído, quedando los *executors* que albergaba fuera del posterior reparto de tareas.

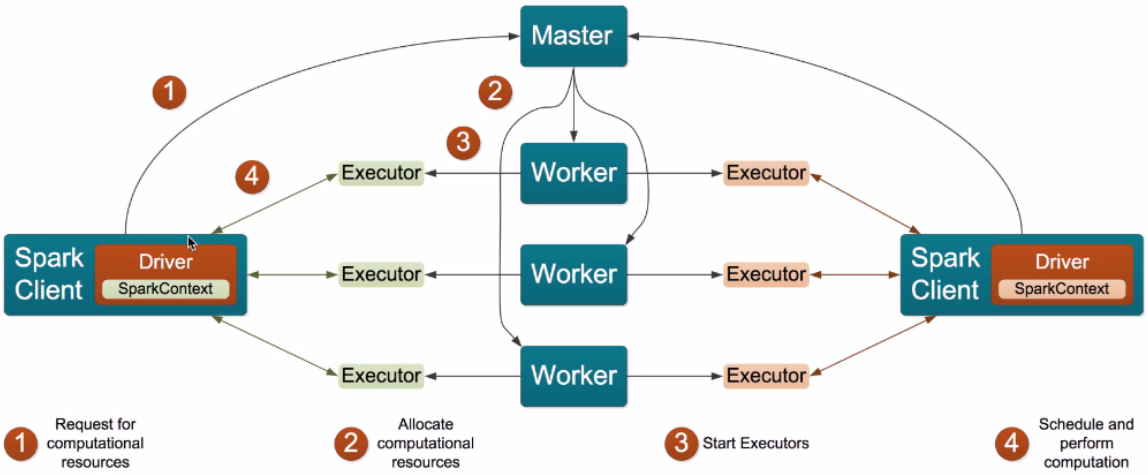


Figura 3: Arquitectura Spark

A la hora de realizar operaciones en Spark, el objeto estrella es el denominado Resilient Distributed Datasets (RDD)[21]. Se trata de una abstracción que mediante diferentes APIs disponibles para Java, Scala y Python permite manipular datos distribuidos por los diferentes nodos del clúster como si estuvieran almacenados de forma local. Este objeto es inmutable, lo cual implica que una vez creado no se le pueden añadir nuevos elementos o eliminar los existentes, solo aplicar transformaciones y acciones sobre el.

Las operaciones que se pueden realizar sobre las RDD se agrupan, tal y como ya se ha adelantado, en transformaciones y acciones. Las primeras transforman un RDD en otro según el criterio indicado y las segundas realizan modificaciones sobre los datos almacenados en dichas RDD. Cabe destacar que las transformaciones en Spark son operaciones

"lazy", lo cual implica que en realidad cada nodo memoriza la secuencia de transformaciones que ha de realizar y los procesa cuando una acción es ejecutada.

Una vez terminada la primera fase en la que los *executors* son creados y enlazados con el *driver*, éste último empieza a analizar la estructura del código y genera un grafo DAG (Directed Acyclic Graph)[19][1] con las operaciones que se realizan sobre la RDD. Partiendo de ese grafo genera un *job* por cada operación de tipo acción que encuentra y dentro de cada *job* separa la ejecución en diferentes *stages* según las dependencias que existan entre operaciones. Por último, cada *stage* es dividido por defecto en unidades de 64MB y a cada unidad resultante se denomina *task*, el cual es enviado a un *executor* para ser procesado. El tamaño de cada *task* puede ser modificado programáticamente, pudiendo de esa forma manipular el número de *task* que un *executor* deba ejecutar. La figura 4 de la página 14 ayuda a visualizar la relación entre los elementos que conforman el proceso de creación de tareas.

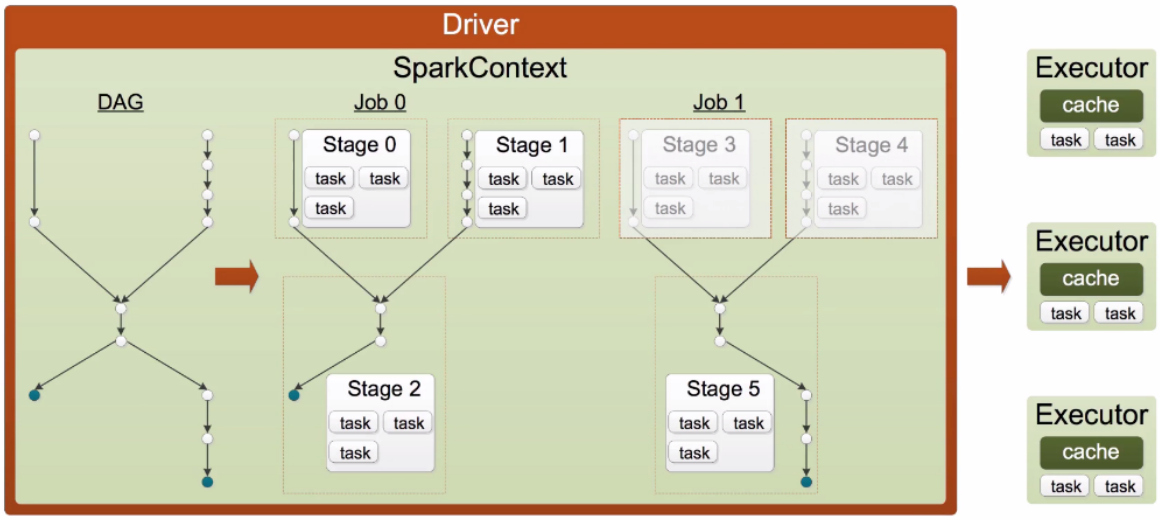


Figura 4: Creación de tareas en Spark

El *driver*, una vez habiendo recibido los resultados de todas las *task* que ha repartido, enviará un mensaje a los *executors* indicando que el procesamiento ha sido finalizado y calcula el resultado final.

Capítulo 3

Entorno de pruebas: Construcción

Conocedores de la problemática a solventar y de las tecnologías que se desean utilizar para ello, es momento de detallar la confección de los entornos de prueba utilizados para medir el rendimiento de dichas tecnologías.

En las siguientes páginas, se describe, por un lado, la construcción de un entorno centralizado compuesto por un único nodo que alberga instancia de MySQL Server, cuyo objetivo es imitar la infraestructura monolítica que impera en Datik. Por el otro lado, se expone el ensamblaje de un entorno distribuido, formado por tres nodos homogéneos sobre los cuales operan Apache Cassandra y Apache Spark en consonancia.

Dichos nodos son erigidos mediante máquinas virtuales que operan sobre distintos dispositivos. Los procesos clientes se alojan en un ordenador portátil mientras que las instancias relativas a los servidores se ejecutan sobre un ordenador de sobremesa, logrando así neutralizar toda diferencia que pudiese existir en cuanto a hardware. Para dar comienzo a este tercer capítulo, se describen las características físicas de los dispositivos recién mencionados.

3.1. Características de las máquinas físicas

Ordenador portátil: Terminal que alberga los procesos cliente encargados de lanzar diversas peticiones contra el servidor y recolectar los resultados computados por este, además de cuantificar el tiempo transcurrido entre ambos eventos.

- Procesador: Intel(R) Core(TM) i-5 4200M CPU @ 2.50 GHz
- Memoria RAM: 8 GB
- Disco Duro: Western Digital WD5000LPCX-24C6HTO SSD 500GB
- Sistema Operativo: Windows 10
- Adaptador Red: Qualcomm Atheros AR8172/8176/8178 PCI-E Fast Ethernet Controller (NDIS 6.30)

Ordenador de sobremesa: Terminal que adquiere el rol de servidor dentro del entramado. Se encarga del almacenamiento y procesamiento de los datos mediante el uso de las tecnología que se analizan en el presente proyecto.

- Procesador: AMD FX(tm)-8350 Eight-Core Processor 4.00GHz
- Memoria RAM: 32 GB
- Disco Duro: Western Digital WD5000AAKX-22ERMA0 SSD 500GB
- Sistema Operativo: Windows 10
- Adaptador Red: Killer e2200 Gigabit Ethernet Controller (NDIS 6.30)

Ambas máquinas se encuentran enlazadas a un *switch* tp-link-tl-sf1008d mediante sendos conectores Fast Ethernet, permitiendo intercambiar datos a una velocidad de 100 Mb/s.

El switch, a su vez, está conectado a un *router* que proporciona acceso al exterior de la red privada, el cual sólo será utilizado para descargar contenido relativo a los preparativos de las pruebas.

3.2. Confección de las máquinas virtuales

Con el objetivo de no realizar las mismas configuraciones reiteradamente, se ha decidido crear una máquina virtual básica siguiendo los pasos que se describen a continuación y tras su puesta a punto, clonarla tantas veces como instancias hagan falta.

1. **Crear la máquina virtual**: Instancia creada mediante VirtualBox utilizando la imagen de Ubuntu Server 16.04 LTS y 75 GB de memoria física. El resto de las características no han sido especificadas ya que es posible reasignar su valor en cualquier momento.

2. **Asignar una IP estática**: La mayoría de los clones generados sobre esta máquina adquieren el rol de servidor, por lo que no es de recibo que su IP pueda variar cada vez que el servicio de red o la máquina misma se reinicien. Para evitar cualquier inconveniente de este estilo, es totalmente necesario configurar una interfaz de red con una IP estática.

3. **Cambiar el adaptador de red a modo puente**: Al crear una máquina virtual con VirtualBox, el adaptador de red predefinido es el NAT. Ello implica que dicha máquina se encontrará aislada en una red lógica y solo podrá acceder a otros dispositivos pasando por la máquina física y haciendo uso de su IP.

   Modificando el adaptador de red al modo puente se consigue extender la red privada hasta el nodo virtual, eliminando así toda dependencia con la máquina física y posibilitando la comunicación con el resto de los dispositivos que residen en la misma red privada.

4. **Instalar Java 8**: Las tecnologías que se desean probar en las futuras pruebas se ejecutan sobre la Máquina Virtual de Java(JVM), por lo que es necesario instalar una versión de Java para su funcionamiento. Entre todas las disponibles se ha optado por la versión 8.

Una vez terminado de confeccionar la máquina virtual básica, es aconsejable ejecutar el comando *tracert* y comprobar ciertas métricas como la latencia y el número de saltos que se dan por la red hasta llegar a otro nodo de la infraestructura. Siguiendo este simple consejo es posible identificar errores relacionados con la configuración y solucionarlos con mayor celeridad.

3.3. Entorno centralizado

El objetivo del entorno centralizado es imitar la infraestructura utilizada por Datik para ejecutar el proceso Cálculo de Indicadores. Dicha infraestructura consta de un servidor en la nube que alberga una instancia de MySQL Server y un proceso que actúa como cliente. Este último realiza consultas contra la base de datos y computa los registros obtenidos para calcular los indicadores deseados.

3.3.1. Servidor

La creación y confección del nodo servidor para el entorno de pruebas centralizado se resume en los siguientes pasos:

1. **Clonar la máquina virtual base**: Crear una copia de la instancia básica confeccionada en el apartado anterior y asignar los parámetros especificados en cuadro 1 de la página 18.

	IP Privada	**Procesadores**	**Memoria RAM**
Nodo MySQL	192.168.0.100	6	24 GB

Cuadro 1: Descripción del nodo centralizado

2. **Descargar e instalar la última versión de MySQL**: Acceder a la máquina virtual recientemente clonada e instalar la última versión de MySQL Server, la 5.7, mediante Advanced Packaging Tool (APT) de Ubuntu.

3. **Crear un esquema**: Identificarse contra MySQL Server utilizando el único usuario existente por el momento, 'root', y la contraseña que se le ha asignado durante la instalación para crear el esquema encargado de almacenar los datos.

4. **Posibilitar acceso desde la máquina remota**: Por motivos de seguridad, las aplicaciones que se conectan a la base de datos no deberían de acceder a ella mediante el usuario 'root', por lo que se recomienda crear un nuevo usuario y especificar los permisos que este ha de tener sobre el esquema creado en el paso anterior.

5. **Abrir los puertos necesarios**: Al estar trabajando sobre máquinas virtuales alojadas en una red propia, todos los puertos se encuentran abiertos por defecto, pero en caso de existir algún cortafuegos que protegiera el servidor, sería necesario abrir el acceso al puerto donde escucha MySQL, el 3306 por defecto.

3.3.2. Emulando el cliente

Emular un proceso cliente en el entorno centralizado es tan simple como instalar la última versión del MySQL Workbench[1] sobre el ordenador portátil (no hace falta máquina virtual alguna) y crear una nueva conexión con el servidor especificando la IP junto al usuario y contraseña correspondientes.

3.4. Entorno distribuido

Infraestructura cuyo núcleo es un clúster compuesto por tres nodos virtuales alojados en el ordenar de sobremesa erigido con el objetivo de almacenar y procesar el dataset de prueba.

Un proceso cliente residente en el ordenar portátil, por su parte, se encarga de ejecutar el programa que especifica el procesamiento que ha de llevar a cabo el clúster y cuantificar el tiempo transcurrido en dicha tarea.

3.4.1. Servidor distribuido

En las siguientes líneas se detalla la configuración de los nodos que conforman el clúster y la puesta a punto de las tecnologías que operan sobre él.

1. **Clonar la máquina virtual base**: Clonar la máquina virtual base tantas veces como nodos vayan a conformar la infraestructura. En éste particular caso se crean 3 copias.

   Además de configurar la IP, el número de procesadores y la memoria RAM disponible para cada uno, es necesario hacer lo propio con el hostname. Apache Cassandra recurre a este atributo para conocer la IP de una máquina cuando ciertos elementos del fichero de configuración no son especificados y en el caso de Spark, el proceso *driver* utiliza los hostname para establecer la conexión con otros nodos de la infraestructura. El cuadro 2 de la página 20 resume la asignación de dichos atributos.

   Cabe destacar que la suma de recursos físicos destinados a la creación del clúster es idéntica a los recursos destinados para erigir el servidor del entorno centralizado.

2. **Resolver hostnames a IPs**: Los hostname recién configurados no sirven de nada si no es posible asociarlos a una IP concreta. Dicho trabajo lo suele realizar un DNS, pero al estar operando sobre una red privada que no dispone de DNS alguno, es

[1] `https://www.mysql.com/products/workbench/`

	IP Privada	**Procesadores**	**Memoria RAM**	**Hostname**
Nodo 1	192.168.0.101	2	8 GB	nodo1
Nodo 2	192.168.0.102	2	8 GB	nodo2
Nodo 3	192.168.0.103	2	8 GB	nodo3

Cuadro 2: Descripción de los nodos que componen el clúster

necesario recurrir al fichero */etc/hosts* para obtener la información deseada. Este fichero debe de albergar la IP privada y el hostname de la propia máquina y del resto de hosts que conforman el clúster.

3. **Descargar Apache Cassandra**: Puede ser instalada mediante paquetes deb o rpm pero en este proyecto se ha optado por descargar el archivo binario y ejecutarlo de forma manual una vez realizadas las configuraciones pertinentes.

   Siendo un proyecto de código abierto relativamente reciente, las nuevas versiones y actualizaciones son lanzadas frecuentemente para arreglar bugs e implementar funcionalidades nuevas. Debido a ello, es imposible trabajar con la última versión, por lo que se ha decidido utilizar la 2.1.8 para el desarrollo del presente proyecto.

4. **Descargar Apache Spark**: Es necesario tener en cuenta la compatibilidad con la versión de Cassandra ya descargada. Aunque en principio se trata de dos proyectos totalmente independientes, existen conectores que buscan facilitar un uso conjunto de estas tecnologías. Analizando la tabla de compatibilidades[2] ofrecida por DataStax, dueña del conector seleccionado para el presente proyecto, se ha decidido descargar la versión 1.4.1 de Spark.

Configuración de Apache Cassandra

El fichero que alberga todos los atributos configurables de Cassandra es el denominado *cassandra.yaml* y se encuentra alojado dentro de la carpeta *conf*. Es necesario modificar en cada uno de los nodos que conforman el clúster, los atributos que se definen a continuación:

- **cluster_name**: Varios clúster lógicos pueden coexistir dentro de uno físico. Éste atributo permite determinar a cual de todos pertenece el nodo.
- **initial_token y num_token**: Al insertar un registro, un proceso denominado *partitioner*[3] se encarga de dilucidar en cual de los nodos se ha de almacenar. Para ello,

[2] `https://github.com/datastax/spark-cassandra-connector`
[3] `https://docs.datastax.com/en/cassandra/2.1/cassandra/architecture/architecturePartitionerAbout_c.html`

cada nodo tiene que conocer de antemano el rango de particiones que se le han atribuido, especificado mediante el atributo initial_token.

No obstante, remover o añadir nuevos nodos en el sistema implica volver a calcular y modificar dicho rango en cada máquina; tarea inviable cuando la envergadura del clúster aumenta considerablemente. Para automatizar las operaciones mencionadas, las versiones más reciente de Cassandra añaden el concepto de nodos virtuales[4] y la opción de activar dicha funcionalidad mediante el atributo num_token.

- **seed_provider**: Lista de direcciones IP delimitadas por coma que se utilizan como punto de contacto cuando un nodo desea unirse al clúster.
- **listen_address**: La dirección IP que utilizan otros nodos para conectarse a una máquina especifica. Si no se indica nada, el nombre de la máquina tiene que conducir a su IP utilizando el fichero *etc/hostname*.
- **rpc_address**: La dirección IP de escucha para conexiones de cliente. Sus valores posibles son:
  - 0.0.0.0: Escucha en todas la interfaces configuradas
  - Dirección IP
  - Nombre de máquina
  - Sin especificar: el nombre de la máquina tiene que conducir a su IP utilizando el fichero *etc/hostname*
- **endpoint_snitch**: Establece el modo en el que Cassandra localiza nodos y envía peticiones de enrutamiento. Todos los nodos han de tener el mismo valor, siendo los siguiente los más utilizados:
  - SimpleSnitch: Se utiliza solo para la implementación de centro de datos únicos.
  - RackInferredSnitch: Determina la ubicación de los nodos por rack o por data center.

Una vez habiendo configurado cada nodo de la forma indica en el cuadro 3 de la página 22, es necesario abrir los puertos detallados en el cuadro 4 de la página 22 para que los procesos Cassandra puedan comunicarse entre los diferentes nodos que conforman el clúster.

[4] `http://docs.datastax.com/en/cassandra/2.1/cassandra/architecture/architectureDataDistributeVnodesUsing_c.html`

Atributo	**Nodo** 1	**Nodo 2**	**Nodo 3**
cluster_name	Test Cluster	Test Cluster	Test Cluster
num_token	256	256	256
seed_provider	192.168.0.102 192.168.0.103	192.168.0.101 192.168.0.103	192.168.0.101 192.168.0.102
listen_address	192.168.0.101	192.168.0.102	192.168.0.103
rpc_address	192.168.0.101	192.168.0.102	192.168.0.103
endpoint_snitch	SimpleSnitch	SimpleSnitch	SimpleSnitch

Cuadro 3: Configuración de los nodos Cassandra

Puerto	**Descripción**
7000	Comunicación entre nodos utilizado en caso de no tener activado TLS
7001	Comunicacion TLS entre nodos
7199	Puerto para acceder a JMX
9042	Puerto nativo de transporte para CQL
9160	Cliente del API Thrift

Cuadro 4: Puertos utilizados por Cassandra

Para finalizar, es necesario arrancar los nodos uno a uno mediante el script *cassandra.sh* que Cassandra mismo facilita dentro de la carpeta */bin* y comprobar que el clúster funciona correctamente utilizando la herramienta *nodetool*[5].

Configuración de Apache Spark

Spark utiliza la memoria RAM de forma intensiva, por lo que se recomienda encarecidamente instalarlo en un clúster totalmente aislado al de Cassandra. No obstante, debido a que los recursos disponibles para lleva adelante el presente proyecto son limitados, no se ha podido seguir la recomendación y Spark ha sido instalado de tal forma que ambas tecnologías comparten nodo.

Al tratarse de un entorno de pruebas donde la cantidad de datos a manipular se encuentra predefinida, la instalación realizada no supone problema alguno y además ofrece diversas ventajas como el no tener que volver a configurar nuevas máquinas dentro del entramado, pudiendo pasar directamente a especificar las configuraciones relativas a Spark:

[5] `http://docs.datastax.com/en/archived/cassandra/2.0/cassandra/tools/toolsNodetool_r.html`

1. **Definir nodos esclavos**: El nodo maestro de Spark necesita saber qué máquinas están dispuestas a ofrecer sus recursos para un futuro procesamiento y poder alojar en ellas los *worker* pertinentes. Para ello es necesario especificar las IP de dichos nodos en el fichero */conf/slaves*. Se ha de crear el directorio en caso de que no exista.

2. **Acceso entre máquinas mediante las claves RSA**: Spark sigue una arquitectura maestro/esclavo donde el maestro necesita conectarse a los esclavos para comunicar las operaciones que estos han de ejecutar. Para automatizar el proceso y evitar tener que introducir la contraseña cada vez que se quiera ejecutar algo, es necesario garantizar el acceso entre máquinas mediante las claves RSA.

   Se recomienda crear los certificados con el comando *ssh-keygen* y distribuirlos a los nodos cuya IP haya sido especificada en el fichero */conf/slaves* mediante el comando *ssh-copy-id usuario@servidor* especificando el usuario que va a ejecutar el proceso Spark.

3. **Resolver hostnames a IPs**: Al instalar Cassandra se ha realizado el presente paso por lo que no hace falta volver a repetirlo. Aún así, cabe recordar que en caso de instalar Spark en un clúster independiente sería necesario manipular el fichero */etc/hosts* para especificar la IP privada y el hostname de la propia máquina y del resto de hosts que conforman el clúster.

4. **Definir el nodo Maestro**: Los nodos esclavos necesitan conocer la IP del terminal que adquiere el rol de maestro para poder aceptar las tareas enviadas por este último. Ello se consigue modificando el fichero *conf/spark-env.sh* y añadiendo las líneas *SPARK_MASTER_HOST=ip_del_host* y *SPARK_LOCAL_IP=ip_del_host* en cada una de ellas.

Antes de iniciar el proceso es necesario cerciorarse de que los puertos mostrados en el cuadro 5 de la página 23 se encuentran abiertos y disponibles para su uso:

Puerto	**Descripción**
8080	Interfaz web para visualizar métricas relativas al maestro
8081	Interfaz web para visualizar métricas relativas a los workers
7077	Conexión con el clúster y envío de tareas
aleatorio	Puerto asignado a cada worker creado para la computación

Cuadro 5: Puertos utilizados por Spark

Para finalizar, solo hace falta lanzar el proceso Spark mediante el script *start-all* disponible en la carpeta *sbin*.

3.4.2. Cliente

Para emular el cliente que interacciona con el clúster del entorno distribuido se ha optado por crear una máquina virtual sobre el ordenador portátil y configurar dentro de dicha instancia los elementos necesarios para lanzar los procesos.

1. **Clonar la máquina virtual base**: Crear una copia de la instancia básica y asignar los parámetros que aparecen en cuadro 6 de la página 24.

	IP Privada	**Procesadores**	**Memoria RAM**
Nodo Cliente	192.168.0.104	2	4 GB

Cuadro 6: Descripción del nodo cliente en el entorno distribuido

2. **Instalar Eclipse**: El entorno de desarrollo integrado seleccionado para ejecutar el proceso cliente es Eclipse Luna.

3. **Instalar Maven**: Herramienta de software para la gestión y construcción de proyectos Java que facilita de sobremanera la gestión de las librerías y sus respectivas dependencias. Instalado mediante Advanced Packaging Tool (APT) de Ubuntu.

4. **Crear un proyecto Java de Maven e incluir dependencias**: Eclipse ofrece la posibilidad de crear un proyecto Java con naturaleza de Maven y así poder insertar las siguientes librerías y sus respectivas dependencias cómodamente:

   - **Cassandra Driver Core** [6]: Contiene los métodos necesarios para interaccionar con Cassandra mediante Java.
   - **Spark Core** [7]: Librería que permite utilizar las funcionalidades centrales de Spark mediante Java.
   - **Connector Spark-Cassandra** [8]: Librería que ofrece funcionalidades para un uso conjunto de Spark y Cassandra

[6] `https://mvnrepository.com/artifact/com.datastax.cassandra/cassandra-driver-core/2.1.8`
[7] `https://mvnrepository.com/artifact/org.apache.spark/spark-core_2.10`
[8] `https://mvnrepository.com/artifact/com.datastax.spark/spark-cassandra-connector_2.10`

Capítulo 4

Almacenamiento y consulta de datos

El almacenamiento y la retribución de los datos constituyen el grueso del presente capítulo. Primero, se expone la naturaleza de los registros que conforman el conjunto de datos y se describen las consultas que se desean realizar sobre este. Después, se presenta el diseño de las tablas utilizadas para almacenar dichos datos, incidiendo especialmente en la correspondiente a Apache Cassandra debido a las peculiaridades que presenta. Para finalizar, se exponen los criterios seguidos a la hora de elegir las consultas ya mencionadas en detrimento a otras posibles candidatas.

4.1. Descripción del Dataset

Para cuantificar los beneficios aportados por las tecnologías propuestas, se ha decidido utilizar un dataset público de aproximadamente 25 GB que recoge la información de los trayectos realizados por los taxis amarillos de Nueva York durante el año 2015[1], ya que los datos actualmente almacenados por Datik se encuentran sujetos a una cláusula de confidencialidad.

El conjunto de datos está compuesto aproximadamente por 146 millones de registros en formato CSV, los cuales se hallan divididos en 12 ficheros de texto, uno por cada mes del año. La tabla 7 de la página 26 ofrece una breve descripción de los atributos que representan un registro o viaje:

[1] `http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml`

VendorID	Código referente proveedor de telefonía que facilita los registros. **1= Creative Mobile Technologies, LLC; 2= VeriFone Inc.**
tpep_pickup_datetime	La fecha y hora en que se activa el medidor.
tpep_dropoff_datetime	La fecha y hora en que se desactiva el medidor.
Passenger_count	Número de pasajeros en el vehículo.
Trip_distance	La distancia de viaje transcurrida en millas según el taxímetro.
Pickup_longitude	Longitud en la que el medidor se activa.
Pickup_latitude	Latitud en la que el medidor se activa.
RateCodeID	El código de tarifa vigente al final del viaje. **1= Tarifa estándar** **2= JFK** **3= Newark** **4= Nassau or Westchester** **5= Precio negociado** **6= Viaje de Grupo**
Store_and_fwd_flag	Indica si el registro refrente al viaje se mantuvo en la memoria del vehículo por falta de conexión con el servidor. **Y= store and forward trip** **N= not a store and forward trip**
Dropoff_longitude	Longitud en la que se desactiva el medidor.
Dropoff_latitude	Latitud en la que se desactiva el medidor.
Payment_type	Código numérico que indica la forma de pago del pasajero. **1= Tarjeta de crédito** **2= Efectivo** **3= Sin cargo** **4= Pleito** **5= Desconocido** **6= Viaje cancelado**
Fare_amount	La tarifa de tiempo y distancia calculada por el contador.
Extra	Varios extras y recargos. Actualmente, sólo incluye los $0.50 de hora punta y $1 de cargos por la noche.
MTA_tax	Impuesto de $0.50 que se activa automáticamente basado en el contador.
Improvement_surcharge	$0.30 de recargo al finalizar el trayecto.
Tip_amount	Éste campo es rellenado automáticamente al pagar con tarjeta de crédito.
Tolls_amount	Importe total de todos los peajes pagados durante el viaje.
Total_amount	La cantidad total cobrada a los pasajeros.

Cuadro 7: Atributos que componen un Viaje

4.1.1. Definición de las consultas

A la hora de diseñar las tablas en cualquier base de datos NoSQL uno de los aspectos a tener en cuenta son las consultas que se van a ejecutar contra ellas. Debido a la naturaleza distribuida que la mayoría poseen de forma inherente, la sintaxis disponible para especificar las consultas se ve drásticamente reducida en pro de una escalabilidad infinita y tiempos de respuesta rápidos.

En la tabla 8 de la página 27 de presentan las consultas que se van a ejecutar contra ambos entornos. El objetivo que se persigue al lanzar cada una de ellas es especificado en el apartado 4.2.3 de este mismo capítulo:

ID Consulta	Descripción
QUERY01	Obtener los datos registrados por la compañía Creative Mobile Technologies en el día 2015-03-10 entre las horas 06:00 y 18:00.
QUERY02	Los 10 días con mayor número de store & foward durante el mes de abril ordenados de forma descendente.
QUERY03	Utilizando los datos de julio y agosto, entre los viajes más largos que la media el porcentaje de pagos que se han realizado con tarjeta de crédito y en efectivo.
QUERY04	Calcular la media aritmética de los atributos passenger_count, trip_distance y total_amount en los viajes de octubre.

Cuadro 8: Descripción de las consultas

4.2. Creación de las tablas

Las consultas que se desean ejecutar contra Cassandra no son el único elemento a tener en cuenta a la hora de crear una tabla. Otro factor de vital relevancia es el tamaño que puede llegar a ocupar una partición, ya que un volumen elevado puede llegar a ralentizar la consulta de sobremanera e incluso impedir que esta pueda finalizarse satisfactoriamente.

MySQL por su parte, no impone restricción alguna, por lo que primero se creará la tabla correspondiente a entorno distribuido, se validará su diseño y después se construirá uno equivalente en la base de datos centralizada.

4.2.1. Entorno distribuido

Análisis de las consultas a realizar

Al igual que en muchos sistemas de almacenamiento, Cassandra utiliza el concepto de clave primaria para identificar de forma unívoca los registros persistidos dentro de una tabla. Dicha clave primaria está compuesta por dos elementos totalmente diferenciados: la *partition key* y el *clustering column*.

Mientras que el *clustring column* solamente sirve para ordenar los datos almacenados en una partición, la *partition key* presenta notorias implicaciones a la hora de insertar registros y consultar los ya almacenados.

Por un lado, al ejecutar una inserción, es utilizada para dilucidar en que nodo del clúster se ha de almacenar el registro. Por otro lado, a la hora de realizar una consulta de lectura, Cassandra obliga a especificar los atributos que conforman la *partition key* para, como su nombre indica, delimitar en qué partición se han de buscar lo datos requeridos y ofrecer de esa manera una respuesta rápida sin importar el volumen de los datos almacenados.

En el caso de operar simplemente con Cassandra, cada tabla debería de estar orientado a responder, de la forma más rápida posible, a una consulta concreta. Esta práctica implica denormalizar los datos que se desean persistir, esto es, almacenar un mismo registro en más de una tabla y lo que es peor, tener que modificar la base de datos cada vez que una nueva consulta es requerida.

En cambio, gracias a herramientas como Spark, toda limitación anteriormente descrita desaparece ya que es suficiente con obtener un subconjunto representativo de los datos para después filtrarlos en memoria. Una de la mayores ventajas que ofrece es la posibilidad de centrarse en la estructura del dataset sin tener que vivir atemorizado por los constantes cambios que pudiesen suceder a nivel de consultas.

Teniendo en cuenta todo lo expuesto y la naturaleza del dataset, parece conveniente crear una tabla cuya *partition key* sea la combinación entre el identificador de la compañía (ofrece la opción de analizar los datos generados por cada compañía y comparar el servicio) y el día del viaje (se quieren calcular los indicadores por día, ergo particionemos los registros por día!). Además, se ha optado por ordenar los registros por la fecha de inicio del servicio, siendo por ende, el atributo *tpep_pickup_datetime* parte del *clustering column*.

Para finalizar, debido a que la combinación de atributos definidos hasta el momento no aseguran que un registro sea unívoco, un nuevo elemento denominado idTrip de tipo UUID[2] ha sido añadido a cada elemento del dataset y definido como parte de la *primary key*.

((vendor_id, dia), tpep_pickup_datetime, idTrip)

Cuadro 9: Primary key de la tabla de Cassandra

Calcular el tamaño de la partición

La definición de un tabla en Cassandra no finaliza al especificar qué atributos conforman la *partition key* correspondiente. Como ya se ha mencionado con anterioridad, las consultas de esta base de datos atacan directamente una partición concreta y en determinadas situaciones un tamaño descontrolado de la misma puede causar que el sistema de almacenamiento no sea capaz de responder a la petición en el tiempo máximo establecido, 10 segundos, haciendo que la consulta falle.

A continuación, se definen unos simples criterios para evitar que, debido a dichas situaciones, las consultas fallen en un futuro:

- **Número de valores**: Una partición ha de tener como mucho 2000 millones de valores almacenados.

  Las columnas que conforman la *partition key* de la tabla, al contrario que las columnas regulares, se almacenan solo una vez en el disco. Por ello, la combinación de las columnas regulares y la cantidad de elementos almacenados es clave para calcular el número de valores de una partición.

  Para obtener el dato del número de registros que puede llegar a almacenar una partición de la tabla, se ha dado por sentado que ambas compañías generan una cantidad de registros sustancialmente parecida.

$$N_r \times (N_c - N_{pk} - N_s) + N_s;$$

[2] `https://en.wikipedia.org/wiki/Universally_unique_identifier`

Nv = Número de valores
Nr = Número de registros
Nc = Número de columnas
Npk = Número de columnas en la Primary Key
Ns = Número de columnas estáticas

Aplicando la fórmula recientemente presentada podemos observar cómo la tabla diseñada respeta holgadamente la restricción de 2.000 millones de valores por partición de Cassandra.

(146.000.000 / 365/2) * (20 - 4 - 0) + 0 = 3.200.000

- **Tamaño de Partición**: El tamaño máximo que puede adquirir una partición no ha de superar unos pocos centenares de MB.

$$\sum_i sizeOf(c_{k_i}) + \sum_j sizeOf(c_{s_j}) + N_r \times \sum_k \left(sizeOf(c_{r_k}) + \sum_l sizeOf(c_{c_l}) \right) + 8 \times N_v$$

sizeof(Cki) = Tamaño de las columnas Partition Key
sizeof(Csj) = Tamaño de las columnas Estáticas
Nr = Número de registros
(sizeof(Crk) + sizeof(Cct)) = Columna clustering + columnas comunes
Nv = Número de valores (Obtenido en la fórmula anterior)

Aplicando la fórmula según los bytes utilizados para representar cada tipo de dato[3]:

sizeof(Cki) = 4 byte + 10 byte = 14 byte
sizeof(Csj) = 0 byte
Nr= 200.000
(sizeof(Crk) + sizeof(Cct)) = ((8+16) + 8) + ((8+16) + 4) + ((8+16) + 8) + ((8+16) + 8) + ((8+16) + 8) + ((8+16) + 4) + ((8+16) + 1) + ((8+16) + 8) + ((8+16) + 8) + ((8+16) + 4) + ((8+16) + 8) + ((8+16) + 8) + ((8+16) +8) + ((8+16) + 8) + ((8+16) + 8) + ((8+16) + 8) + ((8+16) + 8) = 525 byte

[3] `http://docs.datastax.com/en/cql/3.1/cql/cql_reference/cql_data_types_c.html`

8Nv = 8 * 3.200.000 = 25.600.000 byte

Tal y como se puede observar en el recuadro que se presenta a continuación, cada partición de la tabla diseñada albergará a lo sumo 130 MB de datos, volumen aceptable para poder realizar las consultas sobre él sin miedo a recibir error alguno como respuesta.

14 byte + 0 byte + 200.000 * 525 byte + 25.600.000 byte = 130.600.014 byte = 130 MB

Una vez que la tabla diseñada haya superado ambas pruebas es el momento de crearlo físicamente en la base de datos, ya que ofrece todas las garantías necesarias para tener la absoluta certeza sobre su correcto funcionamiento.

4.2.2. Entorno centralizado

Con el objetivo de guardar la máxima similitud con la tabla de Cassandra, se ha optado por utilizar el *primary key* de esta última en MySQL.

Además, a sabiendas de que la consulta MYSQL02 ataca directamente el atributo 'store_fwd' se ha creado un indice secundario sobre este con la idea de acelerar el cómputo.

Tipo	Nombre	Tipo Dato
PK	vendorID	Integer
PK	dia	Varchar
PK	tpep_pickup_datetime	Datetime
PK	tripID	Varchar
IDX	store_fwd	Char
...		

4.2.3. Objetivo de las consultas

Anteriormente se ha mencionado que Cassandra y Spark forman un gran binomio, pero no se ha especificado en ningún momento de qué manera y en qué circunstancias es beneficioso el uso de Spark a la hora de procesar los datos almacenados en Cassandra.

El objetivo de las consultas definidas en el recuadro 8 de la página 27, más allá de obtener información valiosa sobre diferentes eventos, es evidenciar las limitaciones que

presentan las tecnologías propuestas, comprender hasta que punto pueden llegar operando por sí solas y entender cómo se apoyan entre ellas para obtener una mejora sustancial, en comparación a herramientas tradicionales, a la hora de procesar conjuntos hercúleos de datos.

- **QUERY01**: Consulta idílica para Cassandra ya que ataca directamente el contenido de una única partición. Se quiere comprobar que es más rápido obtener datos mediante el sistema de particionado de Cassandra que utilizando los indices de MySQL.
- **QUERY02**: Se desean filtrar los datos mediante una columna que no es parte de la Partion Key, cosa que no es posible en Cassandra y además atacar varias de las particiones existentes. ¿Qué de bien se coordinan las tecnologías analizadas en el presente proyecto para seguir ofreciendo una mejora sustancial de tiempo en situaciones adversas?
- **QUERY03**: Consulta que reutiliza los registros anteriormente cargados en memoria para computar diversas operaciones sobre estos. Se quiere comprobar la veracidad de los supuestos beneficios que Spark ofrece en cómputos de esta índole.
- **QUERY04**: Consulta cuyo único objetivo es calcular varias medias para así simular un funcionamiento simplificado del proceso Cálculo de Indicadores.

Capítulo 5

Procesamiento y Resultados

En el presente capítulo, el penúltimo del documento, se analiza el código implementado para materializar las consultas especificadas en el apartado 4.1.1. A su vez, se publican los resultados obtenidos mediante la ejecución de dichas consultas y a la postre, se detallan las diferencias existentes entre ambos entornos en cuanto a tiempo de ejecución se refiere.

5.1. Procesamiento de datos

El código ejecutado para procesar los datos almacenados en ambos entornos es diametralmente opuesto.

Por un lado, en el entorno centralizado, se hace uso de SQL, un lenguaje de consultas totalmente estandarizado desde hace décadas. Ofrece la posibilidad de recuperar cualquier subconjunto de datos utilizando una sintaxis relativamente sencilla.

Por el otro, en el entorno distribuido, el proceso se complica de sobremanera ya que es necesario lidiar con varias tecnologías a la vez y tener en cuenta las restricciones que cada una de ellas presenta. Ello implica la conveniencia de utilizar una capa de abstracción superior que unifique el funcionamiento de ambas tecnologías, como por ejemplo un conector, el cual dificulta conocer a ciencia cierta lo que se acaba ejecutando en las capas inferiores.

5.1.1. Entorno centralizado

Las consultas SQL lanzadas contra el entorno centralizado no atesoran intríngulis alguna, por lo que se ha optado por omitir los detalle referentes a su sintaxis y presentarlas directamente:

MYSQL01

```
-- ID MYSQL: MYSQL01
-- DESCRIPCIÓN: Obtener los datos registrados por la compañía Creative Mobile
--              Technologies en el día 2015-03-10 entre las horas 06:00 y 18:00

select count(*) from trips where vendor_id = 1
and tpep_pickup_datetime >= '2015-03-10 06:00:00'
and tpep_pickup_datetime <= '2015-03-10 18:00:00'
```

Figura 5: Código referente a la consulta MYSQL01

MYSQL02

```
-- ID MYSQL: MYSQL02
-- DESCRIPCIÓN: Los 10 días con mayor número de store_fwd durante el mes
--              de abril ordenados de forma descendente

select day, count(*) as occurrence from trips where store_fwd = 'Y'
and tpep_pickup_datetime >= '2015-04-01 00:00:00'
and tpep_pickup_datetime < '2015-05-01 00:00:00'
group by day
order by occurrence desc LIMIT 10
```

Figura 6: Código referente a la consulta MYSQL02

MYSQL03

```
-------------------------------------------------------------------------
-- ID MYSQL: MYSQL03
-- DESCRIPCIÓN: Utilizando los datos de julio y agosto, entre los
--              viajes más largos que la media el porcentaje de pagos
--              que se han realizado con tarjeta de crédito y en efectivo
-------------------------------------------------------------------------

create or replace view v1 as select * from trips
where tpep_pickup_datetime >= '2015-07-01 00:00:00'
and tpep_pickup_datetime < '2015-09-01 00:00:00'
and trip_distance > (select avg(trip_distance) from trips
where tpep_pickup_datetime >= '2015-07-01 00:00:00'
and tpep_pickup_datetime < '2015-09-01 00:00:00');

select count(*) from v1 into @long_trips;
select count(*) from v1 where payment_type = 1 into @credit_card;
select count(*) from v1 where payment_type = 2 into @cash;

select (@credit_card/@long_trips) as credit_card_perc,
(@cash/@long_trips) as cash_perc, @long_trips as total_long_trips;
```

Figura 7: Código referente a la consulta MYSQL03

MYSQL04

```
-------------------------------------------------------------------------
-- ID MYSQL : MYSQL04
-- DESCRIPCIÓN: Calcular la media aritmética de los atributos
--              passenger_count, trip_distance y total_amount
--              en los viajes de octubre
-------------------------------------------------------------------------

select avg(passanger_count), avg(trip_distance), avg(total_amount)
from trips  where tpep_pickup_datetime >= '2015-10-01'
and tpep_pickup_datetime < '2015-11-01'
```

Figura 8: Código referente a la consulta MYSQL04

5.1.2. Entorno distribuido

El cuanto al código Java utilizado para representar las consultas del entorno distribuido se refiere, la semántica de su sintaxis ayuda a arrojar luz sobre el uso de cada tecnología, las limitaciones que presentan y la forma en la que se compenetran entre ellas para superar diversas limitaciones.

Por todo ello, al contrario que en el apartado anterior, además de ilustrar el código ejecutado, se ha considerado interesante dedicar unas palabras para explicar el rol que adopta cada tecnología dentro de cada consulta.

CLU01

Ejemplo canónico de una consulta contra Cassandra. Se especifican las *Partition Key* 'vendor_id' y 'dia' para identificar una partición concreta de la tabla y se filtra el contenido de la misma mediante las fechas del *Clustering Column* 'tpep_pickup_datetime'.

```
/*--------------------------------------------------------------------------------
-- ID CLUSTER: CLU01
-- DESCRIPCIÓN: Obtener los datos registrados por la compañía Creative Mobile
--              Technologies en el día 2015-03-10 entre las horas 6:00 y 18:00
--------------------------------------------------------------------------------*/

/* Consultar directamente Cassandra para obtener el resultado */

select count(*) from trips where vendor_id = 1 and dia = '2015-03-10'
and tpep_pickup_datetime >= '2015-03-10 06:00:00'
and tpep_pickup_datetime <= '2015-03-10 18:00:00'
```

Figura 9: Código referente a la consulta CLU01

CLU02

El código que se presenta a continuación trata de exponer un caso en el que Cassandra por si sola no es capaz de satisfacer los requerimientos de una consulta y necesita la ayuda de Spark para conseguir el objetivo.

Debido a que el atributo 'store_fwd' no forma parte de la *Partition Key* definida para la tabla, es totalmente imposible realizar una consulta que filtre los datos mediante dicho criterio. La solución del problema pasa por utilizar Spark, ya que permite almacenar en memoria los registros requeridos y computar las operaciones deseadas libre de toda limitación estructural.

Con el objetivo de no saturar la memoria, se recomienda encarecidamente cargar desde el sistema de almacenamiento solamente el subconjunto de datos que se desea computar. En este caso particular, se quieren obtener los datos de varias compañías a lo largo de diversos días, lo cual implica lanzar una consulta por cada partición de Cassandra y posteriormente parelelizar cada una de las respuestas para que Spark pueda realizar las operaciones pertinentes.

Para simplificar procesos engorrosos como el recientemente descrito, existe la opción de utilizar conectores gratuitos que posibilitan entrelazar el funcionamiento de ambas tecnologías. Tal y como se puede observar en la figura 10 de la página 38, gracias al conector Spark/Cassandra de DataStax[1] basta con crear una lista que contenga tantos objetos como *Partition Key* a consultar y utilizar dicha lista para llamar al método *joinWithCassandraTable*. Este último se encarga de devolver los datos requeridos y cargarlos en memoria de forma totalmente transparente para el usuario.

[1] `https://github.com/datastax/spark-cassandra-connector`

```
/*-------------------------------------------------------------------------------
-- ID CLUSTER: CLU02
-- DESCRIPCIÓN: Los 10 días con mayor número de store_fwd durante el mes
--              de abril ordenados de forma descendente
-------------------------------------------------------------------------------

/* Prepara la lista de Partition keys que se desea consultar en Cassandra. */

Integer[] vendors = {1,2};
List<Integer> vendorList = Arrays.asList(vendors);
List<String> dayList = daysBetweenDates('2015-04-01', '2015-04-31')

/* Utilizando el Spark Context (sc) almacenar en la memoria de todos los nodos
   del cluster las particiones a consultar. */

JavaRDD<TripPK> partitions = sc.parallelize(cartesianProduct(vendorList, dayList));

/* Consultar sobre las distintas particiones de Cassandra y almacenar la respuesta
   en memoria gracias a la función joinWithCassandraTable ofrecido por el conector
   Spark. Se omiten los parametros de mapeo referentes a dicho metodo. */

CassandraJavaPairRDD<TripPK,Trip> rdd =
        javaFunctions(partitions).joinWithCassandraTable(
        "nyt2015", "trips", ...);

/* Computar los datos sobre la memoria RAM de forma distribuida mediante Spark */

List<Trip> result = rdd.filter(t -> t.contains("Y"))
                       .map(t -> t.getDay(), 1)
                       .reduceByKey((a,b) -> (a+b))
                       .sortBy(_._2, false)
                       .take(10)
                       .collect()
                       .foreach(System::println);
```

Figura 10: Código refente a la consulta CLU02

CLU03

La mejor forma de optimizar el procesamiento utilizando Spark es iterar sobre una RDD anteriormente cargada en memoria, ya que esta tecnología es capaz de realizar cómputos resilientes sin tener que almacenar los resultados intermedios en disco.

Por defecto, Spark sólo almacena información en la memoria durante el cómputo, pero ofrece la posibilidad de *cachear* un conjunto de RDDs para no tener que repetir todas las transformaciones realizadas hasta obtener dicho subconjunto de datos. En la figura 11 de la página 39 se puede observar cómo la lista *longTrip* es cacheada para a continuación aplicar tres acciones diferentes sobre ella.

```
/*-------------------------------------------------------------------------------
-- ID CLUSTER: CLU03
-- DESCRIPCIÓN: Utilizando los datos de Julio y Agosto, entre los  viajes más
--              largos que la media el porcentaje de pagos que se han realizado
--              con tarjeta de crédito y en efectivo
-------------------------------------------------------------------------------*/

/* Se cachean los registros obtenidos para poder ejecutar diferentes acciones
   sobre estos sin tener que realizar la misma consulta varias veces */

CassandraJavaPairRDD<TripPK,Trip> rdd =
        javaFunctions(partitions).joinWithCassandraTable(
        "nyt2015", "trips", ...).cache();

/* Obtener la distancia media */

Float tripDistanceMean =  rdd.map(t -> t.getTripDistance()).mean();

/* Filtrar los viajes que superan la distancia media recien calculada */

List<Trip> longTrip = rdd.filter(t -> t.getTripDistance() > tripDistanceMean)
                         .cache();

/* Contar el número de viajes largos, viajes pagados por tarjeta de credito y
   los viajes pagados en metálico */

Integer longTripCount = longTrip.count();
Integer credit = longTrip.filter(t -> t.getPaymentType().equals("1")).count();
Integer cash = longTrip.filter(t -> t.getPaymentType().equals("2")).count();

/* Calcular los porcentajes finales */

System.out.println("Tarj. Crédito": + (credit.toFloat()/longTripCount.toFloat()));
System.out.println("Metálico": + (cash.toFloat()/longTripCount.toFloat()));
```

Figura 11: Código refente a la consulta CLU03

CLU04

Consulta que imita de forma muy simplificada el proceso Cálculo de Indicadores de Ipanel. No aporta nada novedoso en cuanto a las tecnologías se refiere pero refleja la praxis recomendada a la hora de realizar agregaciones: calcular las medias de cada día y almacenarlos en una tabla distinta para más tarde utilizar dichos resultados para calcular los informes semanales, mensuales y anuales.

```
/*-------------------------------------------------------------------------
-- ID CLUSTER: CLU04
-- DESCRIPCIÓN: Calcular la media aritmética de los atributos
--              passenger_count, trip_distance y total_amount
--              en los viajes de octubre
-------------------------------------------------------------------------

/* Se cachean los registros obtenidos para poder ejecutar diferentes acciones
   sobre estos sin tener que realizar la misma consulta varias veces */

CassandraJavaPairRDD<TripPK,Trip> rdd =
        javaFunctions(partitions).joinWithCassandraTable(
        "nyt2015", "trips", ...).cache();

/* Obtener la distancia media la media aritmética de los atributos
   passenger_count, trip_distance y total_amount */

Float passengerCountMean = rdd.filter(t -> t.getPassengerCount()).mean();
Float tripDistMean = rdd.filter(t -> t.getTripDistance()).mean();
Float totalAmountMean = rdd.filter(t -> t.getTotalAmount()).mean();

/* Imprimir los resultados */

System.out.println("Passenger Count Mean: " + passengerCountMean);
System.out.println("Trip Distance Mean: " + tripDistMean);
System.out.println("Total Amount Mean: " + totalAmountMean);
```

Figura 12: Código refente a la consulta CLU04

5.2. Resultados

Una vez habiendo aclarado los entresijos de cada consulta, es hora de poner el broche final al capítulo presentando los resultados obtenidos a consecuencia de su ejecución.

Para realizar dichas consultas ha sido totalmente necesario poblar las bases de datos con anterioridad. Aprovechando esta circunstancia se ha medido el ratio de inserciones por mes ofrecido tanto por MySQL Server como por Cassandra.

5.2.1. Inserciones

Al contrario de lo que se podría esperar inicialmente[15], MySQL Server ofrece un ratio de escritura 15 veces superior al de Cassandra cuando ambas bases de datos se encuentran vacías. No obstante, su rendimiento decae de forma exponencial a medida que se insertan más registros, llegando a rendir peor que Cassandra una vez habiendo insertado la mitad del dataset. Cassandra, por su parte, demuestra ser capaz de mantener un ritmo de inserción constante en todo momento.

La causa principal de estos resultados tan inesperados se pueden resumir en los siguientes tres puntos:

- **Los nodos comparten disco**: Los nodos virtuales que conforman el clúster hacen uso del mismo disco físico, el perteneciente al ordenador de sobremesa. Ello desemboca en un exceso de operaciones entrada/salida que ralentiza las escrituras. La solución pasa por equipar un disco propio a cada nodo de la infraestructura.

- **Las SSTable y el Commitlog comparten disco**: Es posible obtener una mejora aún mayor separando las escrituras de los Commitlog y las SSTable en discos diferentes. Debido a que el acceso a disco en cada caso es totalmente distinto; mientras que los Commitlog se acceden de forma secuencial, las SSTable no siguen un patrón predefinido, tener los dos directorios en el mismo disco puede hacer que las operaciones del cliente se bloqueen entre ellas.

- **Deficiencias del comando copy**: Cassandra ofrece la opción de cargar los datos almacenados en un fichero CSV mediante el comando *copy*[2]. Al contrario que su homologo *load file*[3] de MySQL, el comando *copy* se encuentra diseñado para cargar un número de registros limitados, por lo que su rendimiento decae drásticamente al cargar conjuntos de datos que superen unos pocos miles de registros.

[2] `http://docs.datastax.com/en/cql/3.1/cql/cql_reference/copy_r.html`
[3] `https://dev.mysql.com/doc/refman/5.7/en/load-data.html`

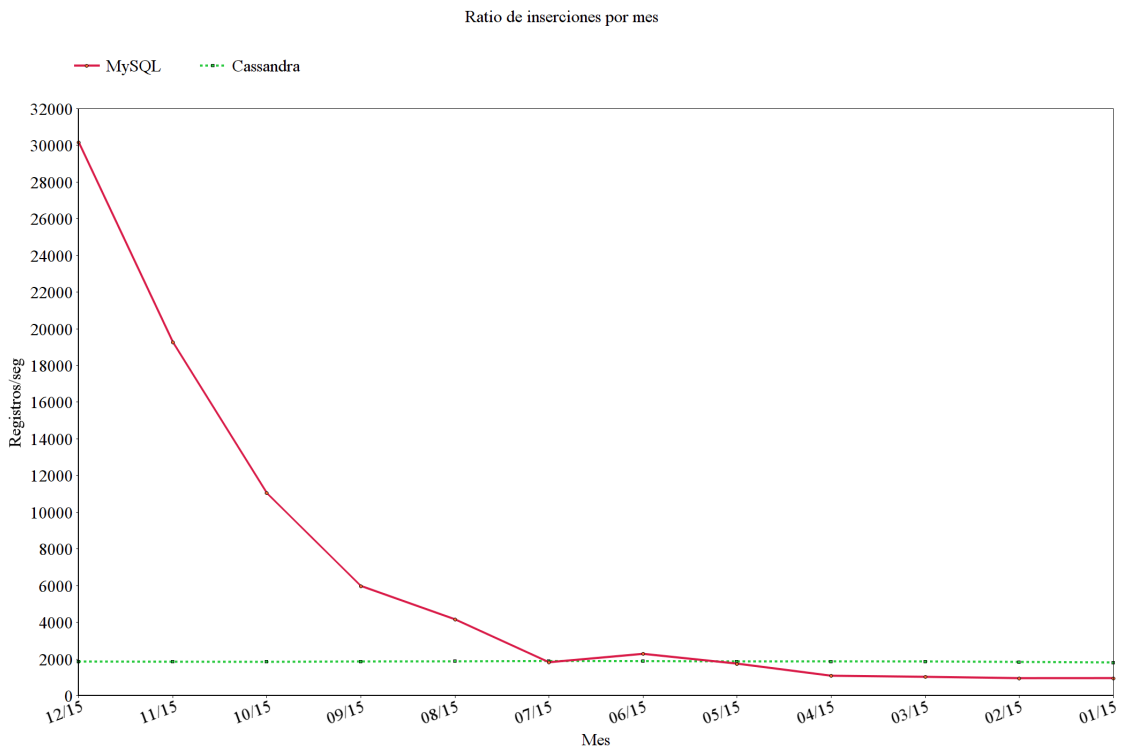


Figura 13: Comparativa tiempos de inserción

Aplicando las optimizaciones recientemente expuestas Cassandra sería netamente superior a MySQL en cuanto al ratio de inserciones/segundo se refiere, pero debido a las características de la máquina física utilizada en el proyecto ha sido totalmente imposible exprimir todo el potencial de esta base de datos distribuida.

5.2.2. Consultas de Lectura

Con el objetivo de medir el rendimiento de cada entorno, las consultas predefinidas en el apartado 4.1.1 de la página 27 han sido ejecutadas 5 veces cada una. En las figuras 14 y 15 se recoge el tiempo transcurrido en el procesamiento de dichas consultas, acompañadas por varios indicadores estadísticos que ofrecen un valor añadido a la información presentada.

MYSQL	MYSQL01	MYSQL02	MYSQL03	MYSQL04
Prueba 1	382,344 s	939,549 s	4271,993 s	1695,817 s
Prueba 2	394,985 s	932,112 s	4289,177 s	1830,015 s
Prueba 3	397,272 s	960,307 s	4746,631 s	1873,289 s
Prueba 4	385,381 s	926,491 s	4801,568 s	1754,751 s
Prueba 5	395,798 s	928,336 s	4112,426 s	1774,333 s
Media	391,156 s	937,359 s	4444,359 s	1785,641 s
Desviación Estándar	6,076 s	12,32 s	276,74 s	61,32 s
Coeficiente Variación	0,015	0,013	0,06	0,034

Figura 14: Resumen de las consultas del entorno centralizado

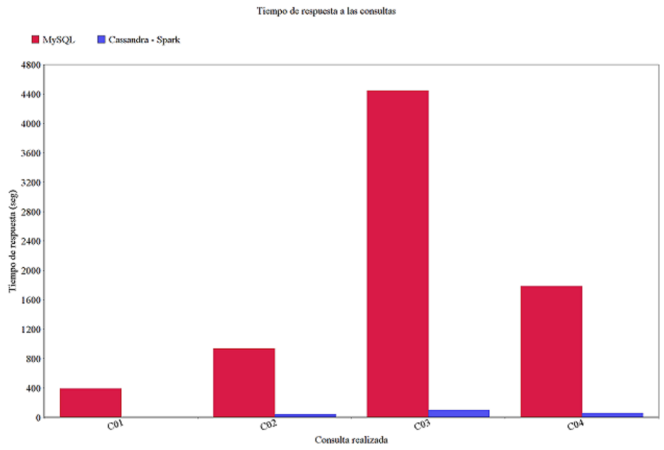
CLUSTER	CLU01	CLU02	CLU03	CLU04
Prueba 1	3,791 s	42,284 s	94,336 s	63,225 s
Prueba 2	3,507 s	43,709 s	101,824 s	59,091 s
Prueba 3	3,621 s	38,531 s	107,354 s	60,113 s
Prueba 4	4,048 s	40,997 s	97,798 s	56,589 s
Prueba 5	3,673 s	38,029 s	101,688 s	54,982 s
Media	3,728 s	40,71 s	100,6 s	58,8 s
Desviación Estándar	0.18 s	2.17 s	4.37 s	2.85 s
Coeficiente Variación	0.04	0.05	0.04	0.048

Figura 15: Resumen de las consultas del entorno distribuido

Tal y como se ha podido comprobar, la desviación estándar es inferior al 10 % en todos los casos, lo cual implica que la diferencia entre los tiempos obtenidos es muy pequeña, legitimando así el valor de las medias calculadas.

En la figura 16 de la página 44 se vislumbra de forma gráfica la inmensa mejora obtenida mediante el entorno distribuido en cuanto a tiempo de ejecución se refiere.

En el peor de los casos el ahorro de tiempo es superior al 95 %, 20 o más veces rápido que su consulta homóloga en el entorno centralizado, dejando en clara evidencia la mejora que ofrece el uso conjunto de Cassandra y Spark a la hora de computar volúmenes masivos de datos.



ID Consulta	MySQL	Cassandra/Spark	Ahorro
QUERY01	391,156 seg	3,728 seg	99,05%
QUERY02	937,359 seg	40,710 seg	95,66%
QUERY03	4444,407 seg	100,6 seg	97,74%
QUERY04	1785,641 seg	58,8 seg	96,71%

Figura 16: Comparativa entre los resultados de ambos entornos

Capítulo 6

Conclusiones

Tal y como se ha podido apreciar durante el transcurso de la presente tesina, muchas son las ventajas ofrecidas por el uso conjunto de Apache Cassandra y Apache Spark. Entre ellas, cabe resaltar la posibilidad de operar sin punto único de fallo, escalar tanto en almacenamiento como procesamiento prácticamente hasta el infinito y además, mejorar de forma inequívoca, los tiempos de respuesta ofrecidos por MySQL cuando se opera sobre un dataset de volumen considerable.

No obstante, existen una seria de inconvenientes a la hora de empezar a gozar de las ventajas descritas en el anterior párrafo.

Utilizar este tipo de tecnologías supone un cambio de paradigma enorme en la forma de tratar la información y por ende requiere una inversión de tiempo considerable en la formación del personal y en migrar la arquitectura ya existente.

Si de antemano el cambio de paradigma puede repeler a usuarios encallados en bases de datos tradicionales que por diversas circunstancias no encuentran el momento idóneo para dar el salto, la inmadurez de las tecnologías utilizadas en el entorno distribuido no ayuda mucho a disipar las dudas. Al tratarse de software creado recientemente, las actualizaciones son constantes y es difícil mantener una versión estable en entornos de producción.

Siguiendo al hilo del mantenimiento, a día de hoy no existe herramienta gratuita alguna que facilite el mantenimiento del clúster, por lo que si no se desea pagar suma elevada de dinero a compañías que ofrecen dicho servicio, es totalmente necesario contar con personal cualificado en mantenimiento de sistemas, algo inviable para empresas pequeñas.

En un futuro muy cercano se augura que los inconvenientes recién descritos se irán disipando y que todo tipo de usuario será capaz de sacar provecho a las incuestionables ventajas que ofrecen tanto las tecnologías analizas en presente proyecto, como sus homólogas. No cabe duda alguna que el Big Data es la revolución del presente que impulsa la evolución hacía el futuro.

Bibliografía y Referencias

[1] Jørgen Bang-Jensen. “2.1 Acyclic Digraphs”. En: *Digraphs: Theory, Algorithms and Applications* (2008), págs. 32-34.

[2] Tushar Deepak Chandra y Sam Toueg. “Unreliable failure detectors for reliable distributed systems”. En: *Journal of the ACM (JACM)* 43.2 (1996), págs. 225-267.

[3] Fay Chang y col. “Bigtable: A distributed storage system for structured data”. En: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), pág. 4.

[4] Jeffrey Dean y Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. En: *Communications of the ACM* 51.1 (2008), págs. 107-113.

[5] Giuseppe DeCandia y col. “Dynamo: amazon’s highly available key-value store”. En: *ACM SIGOPS Operating Systems Review* 41.6 (2007), págs. 205-220.

[6] Alan Demers y col. “Epidemic algorithms for replicated database maintenance”. En: *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. ACM. 1987, págs. 1-12.

[7] Seth Gilbert y Nancy Lynch. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”. En: *ACM SIGACT News* 33.2 (2002), págs. 51-59.

[8] Marcel JE Golay. “Notes on digital coding”. En: *Proceedings of the Institute of Radio Engineers* 37.6 (1949), págs. 657-657.

[9] Naohiro Hayashibara y col. “The/spl phi/accrual failure detector”. En: *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*. IEEE. 2004, págs. 66-78.

[10] Avinash Lakshman y Prashant Malik. “Cassandra: a decentralized structured storage system”. En: *ACM SIGOPS Operating Systems Review* 44.2 (2010), págs. 35-40.

[11] James Manyika y col. “Big data: The next frontier for innovation, competition, and productivity”. En: (2011).

[12] Ralph C Merkle. "A digital signature based on a conventional encryption function". En: *Conference on the Theory and Application of Cryptographic Techniques*. Springer. 1987, págs. 369-378.

[13] Raghunath Nambiar. "Towards an Industry Standard for Benchmarking Big Data Workloads". En: ().

[14] Bruce Jay Nelson. *Remote procedure call*. Inf. téc. Carnegie-Mellon Univ. Dept. Comput. Sci., 1981.

[15] Tilmann Rabl y col. "Solving big data challenges for enterprise application performance management". En: *Proceedings of the VLDB Endowment* 5.12 (2012), págs. 1724-1735.

[16] Alexandru Dan SICOE y col. *A Persistent Back-End for the ATLAS Online Information Service (P-BEAST)*. Inf. téc. ATL-COM-DAQ-2012-010, 2012.

[17] Mark Slee, Aditya Agarwal y Marc Kwiatkowski. "Thrift: Scalable cross-language services implementation". En: *Facebook White Paper* 5.8 (2007).

[18] *The State of Big Data in the Large Corporate World*. Big Data Executive Survey 2013 Boston. 2013.

[19] K Thulasiraman y MNS Swamy. "5.7 Acyclic Directed Graphs". En: *Graphs: Theory and Algorithms* (1992), pág. 118.

[20] Reynold S Xin y col. "Shark: SQL and rich analytics at scale". En: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*. ACM. 2013, págs. 13-24.

[21] Matei Zaharia y col. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". En: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, págs. 2-2.

[22] Matei Zaharia y col. "Spark: cluster computing with working sets." En: *HotCloud* 10 (2010), págs. 10-10.